Exact String Matching

Professor Sean Goggins

Examples from "geeksforgeeks.org"

The Z Algorithm

- This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is O(m + n) with linear space complexity. Now we can see that both time and space complexity is same as KMP algorithm but this algorithm is Simpler to understand.
- In this algorithm, we construct a Z array.

What is Z Array?

- For a string str[0..n-1], Z array is of same length as string. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of Z array is meaning less as complete string is always prefix of itself.

```
Example:
Index          0   1   2   3   4   5   6   7   8   9  10  11
Text           a   a   b   c   a   a   b   x   a   a   a   z
Z values       X   1   0   0   3   1   0   0   2   2   1   0
```

```
More Examples:
str  = "aaaaaa"
Z[]  = {x, 5, 4, 3, 2, 1}

str = "aabaacd"
Z[] = {x, 1, 0, 2, 1, 0, 0}

str = "abababab"
Z[] = {x, 0, 6, 0, 4, 0, 2, 0}
```

- How is Z array helpful in Searching Pattern in Linear time?
    - The idea is to concatenate pattern and text, and create a string "P$T" where P is pattern, $ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

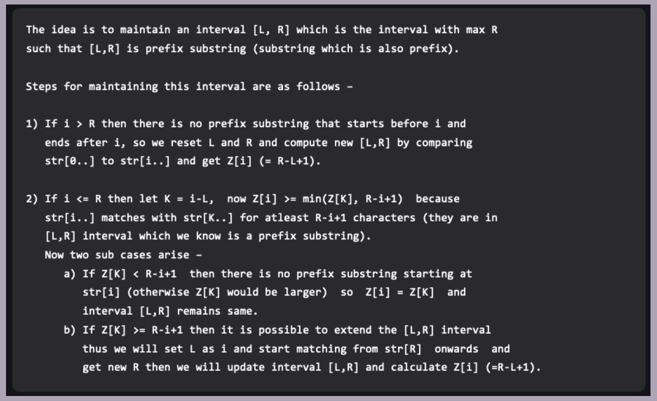
```
Example:
Pattern P = "aab",  Text T = "baabaa"

The concatenated string is = "aab$baabaa"

Z array for above concatenated string is {x, 1, 0, 0, 0,
                                          3, 1, 0, 2, 1}.
Since length of pattern is 3, the value 3 in Z array
indicates presence of pattern.
```

How to construct Z array?

- A Simple Solution is to run two nested loops, the outer loop goes to every index and the inner loop finds length of the longest prefix that matches the substring starting at the current index. The time complexity of this solution is O(n2).
- We can construct Z array in linear time.



```
The idea is to maintain an interval [L, R] which is the interval with max R
such that [L,R] is prefix substring (substring which is also prefix).

Steps for maintaining this interval are as follows –

1) If i > R then there is no prefix substring that starts before i and
   ends after i, so we reset L and R and compute new [L,R] by comparing
   str[0..] to str[i..] and get Z[i] (= R-L+1).

2) If i <= R then let K = i-L,  now Z[i] >= min(Z[K], R-i+1)  because
   str[i..] matches with str[K..] for atleast R-i+1 characters (they are in
   [L,R] interval which we know is a prefix substring).
   Now two sub cases arise –
      a) If Z[K] < R-i+1  then there is no prefix substring starting at
         str[i] (otherwise Z[K] would be larger)  so  Z[i] = Z[K]  and
         interval [L,R] remains same.
      b) If Z[K] >= R-i+1 then it is possible to extend the [L,R] interval
         thus we will set L as i and start matching from str[R]  onwards  and
         get new R then we will update interval [L,R] and calculate Z[i] (=R-L+1).
```

Knuth Morris Pratt Algorithm

- Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[].
- You may assume that n > m.

- Examples:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```

Boyer-Moore Algorithm

- Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

- Examples:

  Input: txt[] = "THIS IS A TEST TEXT"

  pat[] = "TEST"

  Output: Pattern found at index 10

  Input: txt[] = "AABAACAADAABAABA"

  pat[] = "AABA"

  Output: Pattern found at index 0

  Pattern found at index 9

  Pattern found at index 12

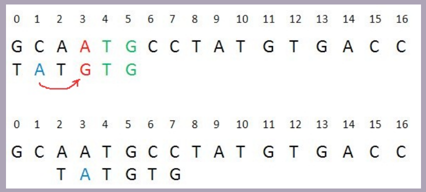
Boyer Moore algorithm also preprocesses the pattern.
Boyer Moore is a combination of the following two approaches.
1) Bad Character Heuristic
2) Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. So it uses greatest offset suggested by the two heuristics at every step.

Unlike the previous pattern searching algorithms, the Boyer Moore algorithm starts matching from the last character of the pattern.

Bad Character Heuristic

The idea of bad character heuristic is simple. The character of the text which doesn’t match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –
1) The mismatch becomes a match
2) Pattern P moves past the mismatched character.

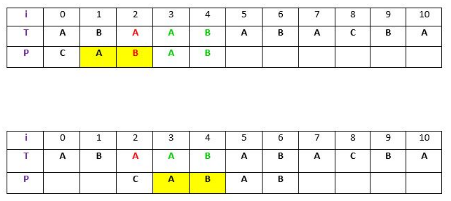


In the above example, we got a mismatch at position 3. Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

Good Suffix Heuristic

Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :
1) Another occurrence of t in P matched with t in T.
2) A prefix of P, which matches with suffix of t
3) P moves past t

Case 1: Another occurrence of t in P matched with t in T
Pattern P might contain few more occurrences of **t**. In such case, we will try to shift the pattern to align that occurrence with t in text T. For example-



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

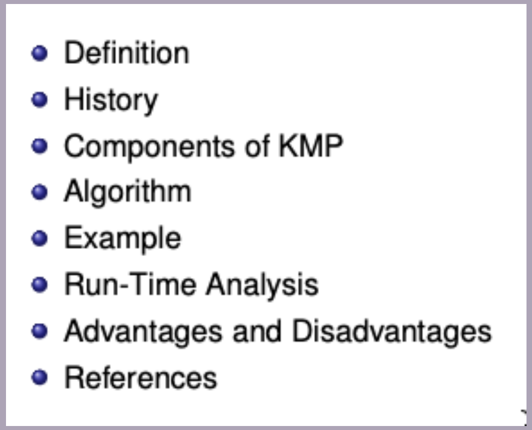
i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

.

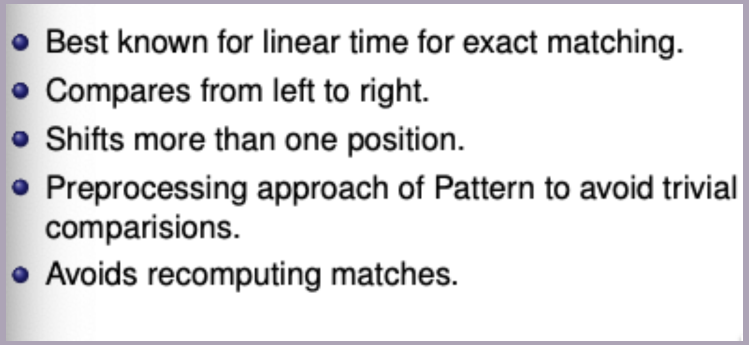
In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2. Now we will search for occurrence of t (“AB”) in P. We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align t in P with t in T. This is weak rule of original Boyer Moore and not much effective, we will discuss a **Strong Good Suffix rule** shortly.

Knuth Morris Pratt Algorithm



- Definition
- History
- Components of KMP
- Algorithm
- Example
- Run-Time Analysis
- Advantages and Disadvantages
- References

Knuth Morris Pratt Algorithm

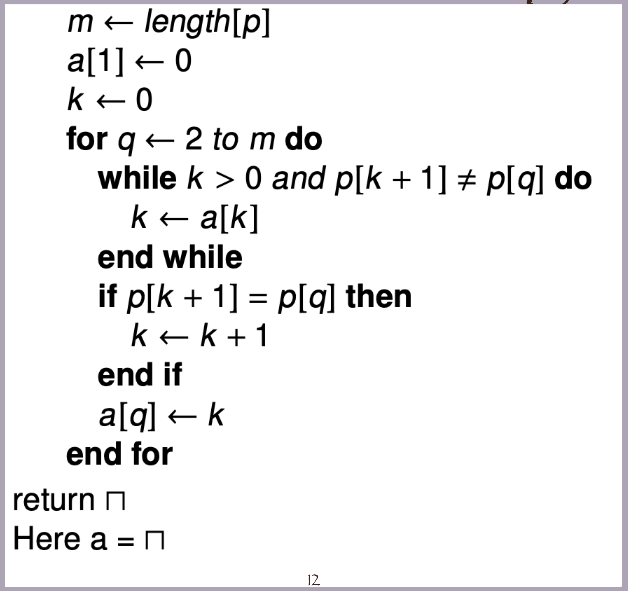


- Best known for linear time for exact matching.
- Compares from left to right.
- Shifts more than one position.
- Preprocessing approach of Pattern to avoid trivial comparisions.
- Avoids recomputing matches.

Knuth Morris Pratt Algorithm

- The prefix-function ⊓ :
  ★ It preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
  ★ It is defined as the size of the largest prefix of $P[0..j-1]$ that is also a suffix of $P[1..j]$.
  ★ It also indicates how much of the last comparison can be reused if it fails.
  ★ It enables avoiding backtracking on the string '*S*'.

Knuth Morris Pratt Algorithm



```
    $m \leftarrow length[p]$
    $a[1] \leftarrow 0$
    $k \leftarrow 0$
    **for** $q \leftarrow 2$ *to* $m$ **do**
      **while** $k > 0$ *and* $p[k+1] \neq p[q]$ **do**
        $k \leftarrow a[k]$
      **end while**
      **if** $p[k+1] = p[q]$ **then**
        $k \leftarrow k + 1$
      **end if**
      $a[q] \leftarrow k$
    **end for**
return ⊓
Here a = ⊓
```

Knuth Morris Pratt Algorithm

- Let us consider an example of how to compute ⊓ for the pattern ‘*p*’.

Pattern	a	b	a	b	a	c	a

Initially : m = length[p]= 7
⊓[1]= 0
k=0

where m, ⊓[1], and k are the length of the pattern, prefix function and initial potential value respectively.

Knuth Morris Pratt Algorithm

Step 1: q = 2, k = 0
⊓[2]= 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
⊓	0	0					

Step 2: q = 3, k = 0
⊓[3]= 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
⊓	0	0	1				

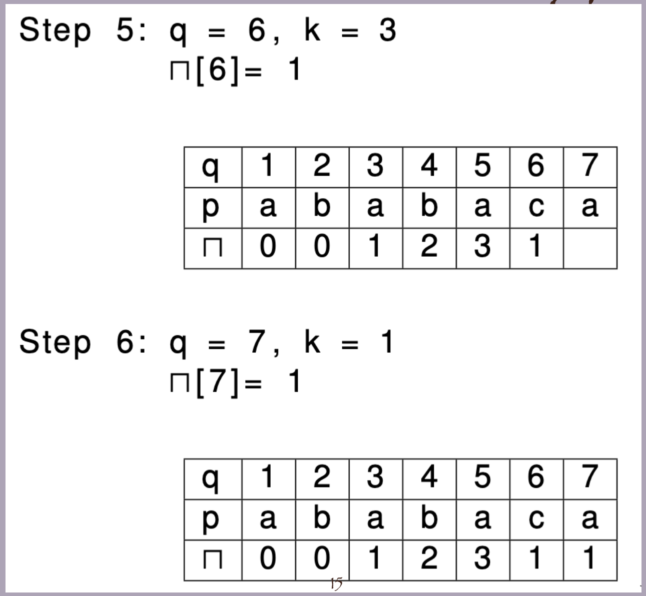
Step 3: q = 4, k = 1
⊓[4]= 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
⊓	0	0	1	2			

Step 4: q = 5, k = 2
⊓[5]= 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
⊓	0	0	1	2	3		

Knuth Morris Pratt Algorithm



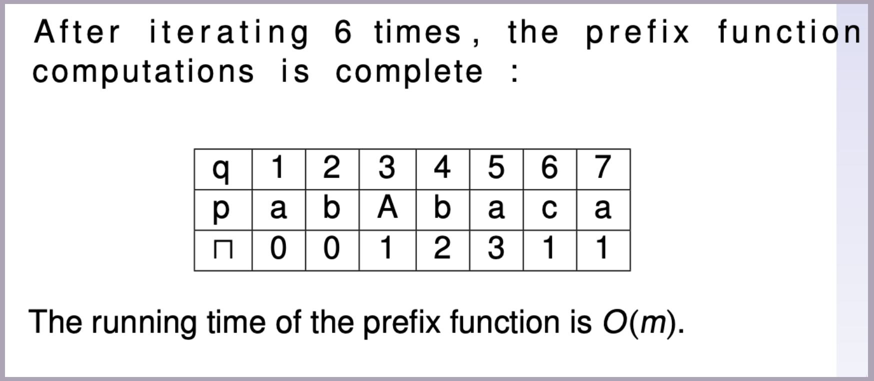
Step 5: q = 6, k = 3
⊓[6]= 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
⊓	0	0	1	2	3	1	

Step 6: q = 7, k = 1
⊓[7]= 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
⊓	0	0	1	2	3	1	1

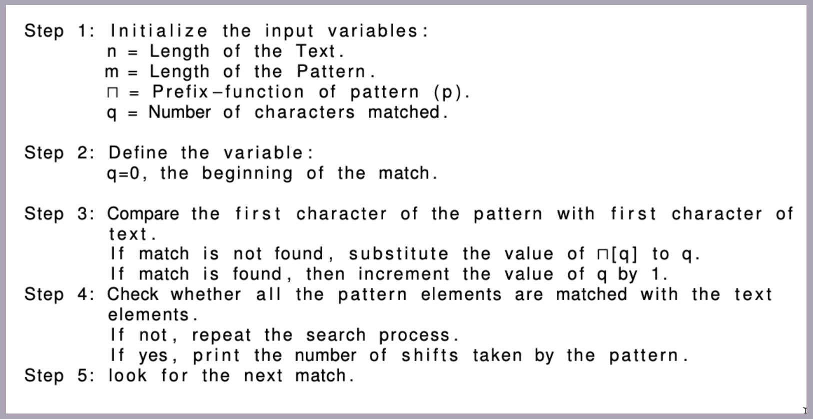
Knuth Morris Pratt Algorithm



After iterating 6 times, the prefix function computations is complete :

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
⊓	0	0	1	2	3	1	1

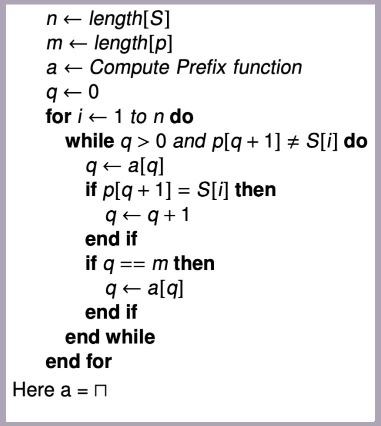
The running time of the prefix function is $O(m)$.



```
Step 1: Initialize the input variables:
        n = Length of the Text.
        m = Length of the Pattern.
        ⊓ = Prefix-function of pattern (p).
        q = Number of characters matched.

Step 2: Define the variable:
        q=0, the beginning of the match.

Step 3: Compare the first character of the pattern with first character of
        text.
        If match is not found, substitute the value of ⊓[q] to q.
        If match is found, then increment the value of q by 1.
Step 4: Check whether all the pattern elements are matched with the text
        elements.
        If not, repeat the search process.
        If yes, print the number of shifts taken by the pattern.
Step 5: look for the next match.
```



```
    $n \leftarrow length[S]$
    $m \leftarrow length[p]$
    $a \leftarrow Compute\ Prefix\ function$
    $q \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        while $q > 0$ and $p[q+1] \neq S[i]$ do
            $q \leftarrow a[q]$
            if $p[q+1] = S[i]$ then
                $q \leftarrow q + 1$
            end if
            if $q == m$ then
                $q \leftarrow a[q]$
            end if
        end while
    end for
Here a = $\Pi$
```

Example of KMP Algorithm

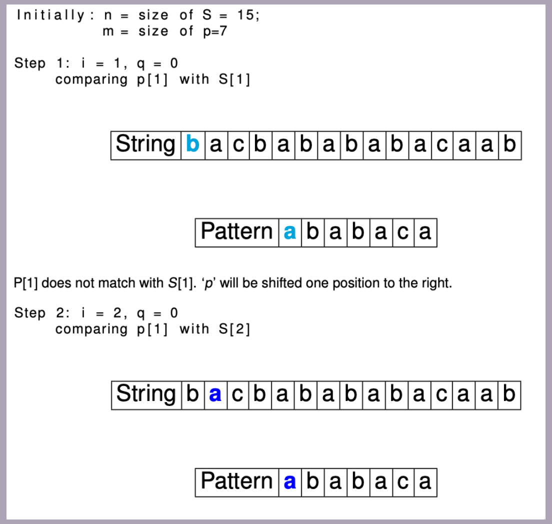
Now let us consider an example so that the algorithm can be clearly understood.

String	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Pattern	a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether ‘*p*’ occurs in ‘*S*’.

Initially…



```
Initially: n = size of S = 15;
           m = size of p=7

Step 1: i = 1, q = 0
     comparing p[1] with S[1]
```

String	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Pattern	a	b	a	b	a	c	a

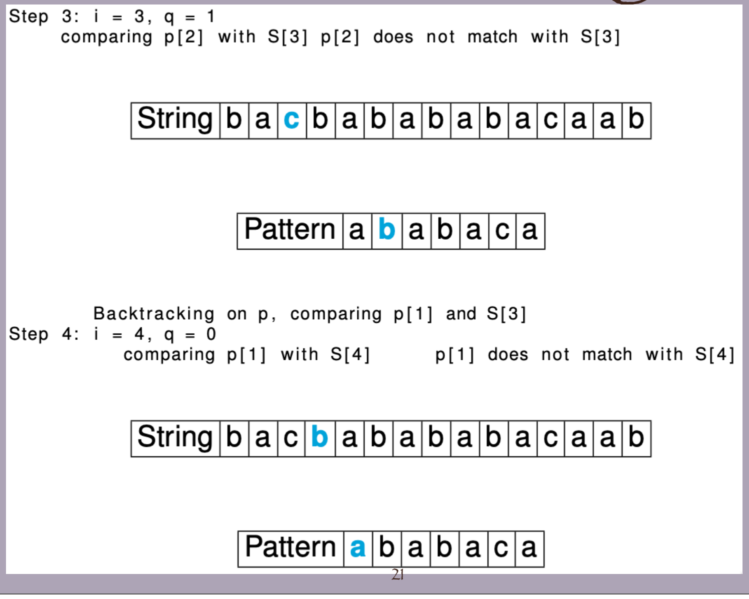
P[1] does not match with *S*[1]. '*p*' will be shifted one position to the right.

```
Step 2: i = 2, q = 0
     comparing p[1] with S[2]
```

String	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Pattern	a	b	a	b	a	c	a

Knuth Morris Pratt Algorithm



```
Step 3: i = 3, q = 1
     comparing p[2] with S[3] p[2] does not match with S[3]
```

String	b	a	**c**	b	a	b	a	b	a	b	a	c	a	a	b

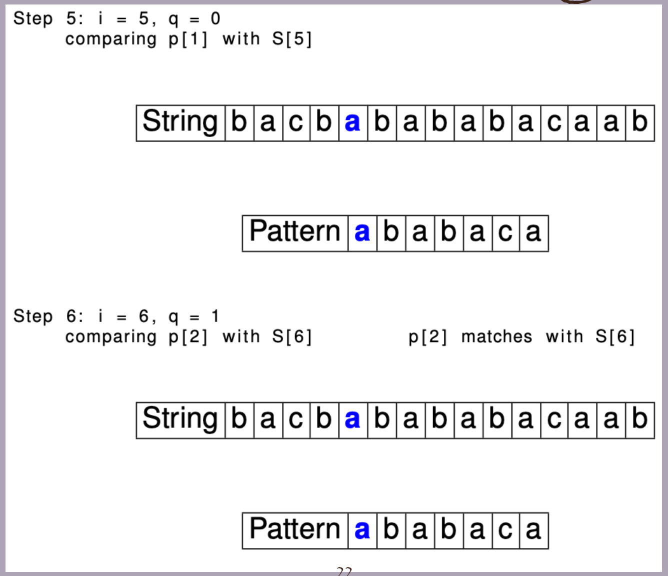
Pattern	a	**b**	a	b	a	c	a

```
        Backtracking on p, comparing p[1] and S[3]
Step 4: i = 4, q = 0
           comparing p[1] with S[4]      p[1] does not match with S[4]
```

String	b	a	c	**b**	a	b	a	b	a	b	a	c	a	a	b

Pattern	**a**	b	a	b	a	c	a

Knuth Morris Pratt Algorithm



```
Step 5: i = 5, q = 0
        comparing p[1] with S[5]
```

String	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

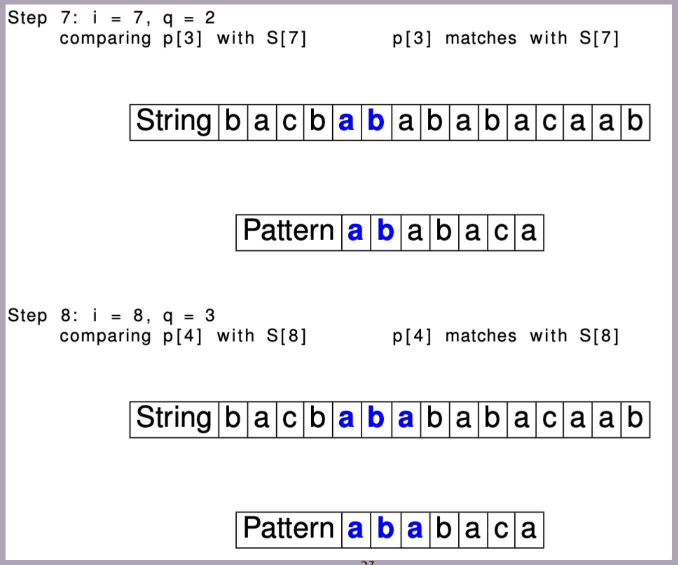
Pattern	a	b	a	b	a	c	a

```
Step 6: i = 6, q = 1
        comparing p[2] with S[6]          p[2] matches with S[6]
```

String	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Pattern	a	b	a	b	a	c	a

Knuth Morris Pratt Algorithm



```
Step 7: i = 7, q = 2
     comparing p[3] with S[7]        p[3] matches with S[7]
```

String	b	a	c	b	**a**	**b**	a	b	a	b	a	c	a	a	b

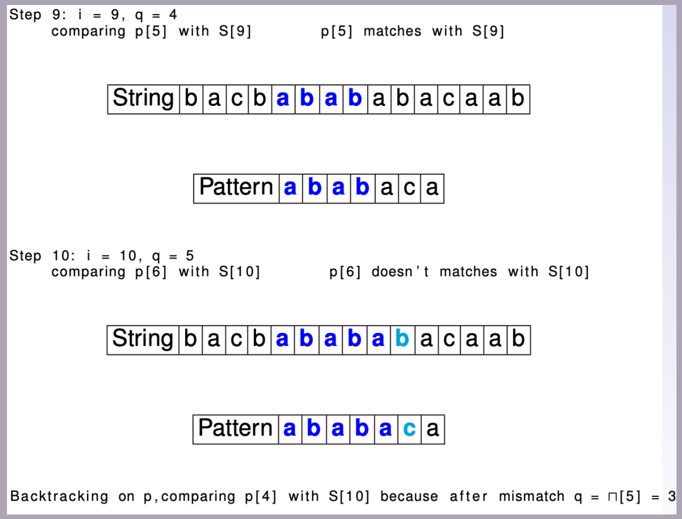
Pattern	**a**	**b**	a	b	a	c	a

```
Step 8: i = 8, q = 3
     comparing p[4] with S[8]        p[4] matches with S[8]
```

String	b	a	c	b	**a**	**b**	**a**	b	a	b	a	c	a	a	b

Pattern	**a**	**b**	**a**	b	a	c	a

Knuth Morris Pratt Algorithm



```
Step 9: i = 9, q = 4
     comparing p[5] with S[9]        p[5] matches with S[9]
```

String	b	a	c	b	**a**	**b**	**a**	**b**	a	b	a	c	a	a	b

Pattern	**a**	**b**	**a**	**b**	a	c	a

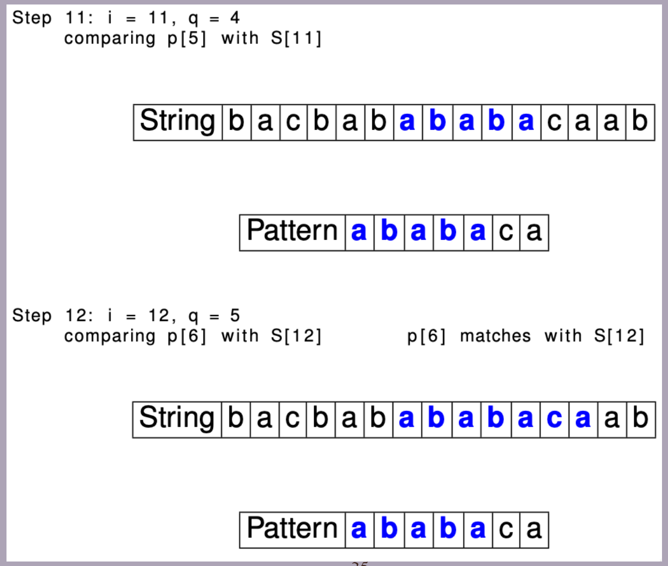
```
Step 10: i = 10, q = 5
     comparing p[6] with S[10]        p[6] doesn't matches with S[10]
```

String	b	a	c	b	**a**	**b**	**a**	**b**	**a**	**b**	a	c	a	a	b

Pattern	**a**	**b**	**a**	**b**	**a**	**c**	a

Backtracking on p,comparing p[4] with S[10] because after mismatch q = π[5] = 3

Knuth Morris Pratt Algorithm



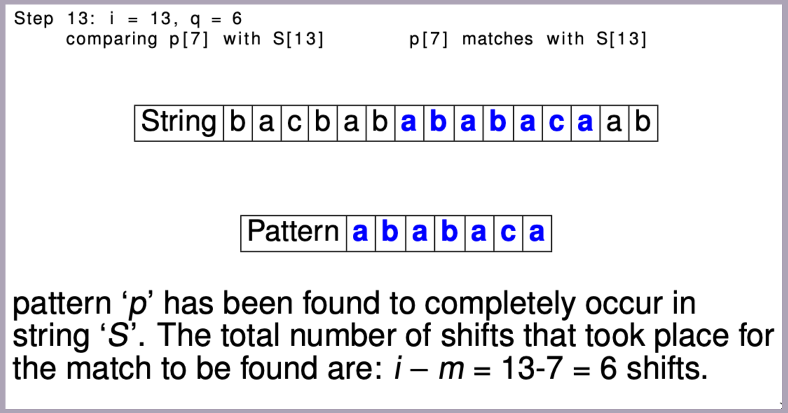
```
Step 11: i = 11, q = 4
      comparing p[5] with S[11]
```

String b a c b a b a b a b a c a a b
Pattern a b a b a c a

```
Step 12: i = 12, q = 5
      comparing p[6] with S[12]          p[6] matches with S[12]
```

String b a c b a b a b a b a c a a b
Pattern a b a b a c a

Knuth Morris Pratt Algorithm



Step 13: i = 13, q = 6
comparing p[7] with S[13]   p[7] matches with S[13]

String	b	a	c	b	a	b	**a**	**b**	**a**	**b**	**a**	**c**	**a**	a	b

Pattern	**a**	**b**	**a**	**b**	**a**	**c**	**a**

pattern ‘*p*’ has been found to completely occur in string ‘*S*’. The total number of shifts that took place for the match to be found are: $i - m$ = 13-7 = 6 shifts.